SWT - CA1

Fereshte Bagheri - Mohammad Amanlou

810100089 - 810100084

https://github.com/MohammadAmanlou/SWT-Fall03

5a6bdc9030fe5a6833a311f7dc05f68a0db100a6

سوال اول - روش های Assume

این متد ها برای تعریف شرایطی استفاده می شود که یک تست در آن معنادار باشد. برای مثال اگر برای ران شدن تستی نیازی به پیش نیاز داشته باشیم، با استفاده از این متد ها، این شرط را چک می کنیم و اگر شرایط درست نباشد، تست halt می شود. halt شدن تست به معنی fail شدن آن نیست و فقط به دلیل برقرار نبودن شرایط اولیه، ignore می شود. این متد ها در ساختار هایی مانند Theory برای در نظر نگرفتن تست های بی معنا استفاده می شود. در واقع این متد ها فقط تحت شرایط خاصی باید اجرا شوند. و کاربرد اصلی آن ها در unit test ها است تا اجرای آزمون را مشروط بر یکسری از شرایط کنند که اگر برقرار نباشد به جای شکست خوردن آزمون آن آزمون نادیده گرفته می شود.

Assume True: اگر این تابع بر روی یک expression با مقدار false صدا شود، چون شرایط اولیه تست برقرار نبوده، تست halt شده و انجام نمی شود. درر واقع اگر شرط داده‌شده درست باشد، آزمون ادامه پیدا می‌کند. اگر شرط نادرست باشد، تست لغو می‌شود و نادیده گرفته می‌شود.

سوال دوم - Thread Safety

برای بررسی Thread Safety، باید شرایط پیچیده‌ای از اجرای همزمان چندین Thread و دسترسی به داده‌های مشترک را شبیه‌سازی کنیم. Unit test ها بطور کلی برای این کار طراحی نشده‌اند. استفاده از آزمون واحد برای بررسی کد های multi-thread مناسب نیست، چون این تست ها برای بررسی عملکرد یک متد یا تابع و به صورت مستقل از سایر اجزا استفاده می شود و مشکلاتی مثل race condition یا deadlock به زمان‌بندی اجرای تردها بستگی داره و این زمان‌بندی هر بار ممکن است متفاوت باشد. اما در بررسی مشکلات مرتبط با thread safety، نیاز داریم تا شرایط concurrency را شبیه سازی کنیم. یعنی باید حالتی را شبیه سازی کنیم که همزمان چند thread به یک منبع مشترک دسترسی دارند. از آنجا که در آزمون واحد به دلیل وجود تنها یک thread، قادر به اجرای موازی کد نیستیم، نمی توانیم این شرایط را شبیه سازی کنیم پس قادر به بررسی رفتار یک برنامه در هنگام deadlock، race condition یا starvation نیستیم.

راه حل ها:

1. تست‌های thread-based: یعنی تست‌هایی بنویسیم که چندین ترد همزمان روی منابع مشترک کار کنند. اینجوری اگر مشکلی باشد، احتمالاً مشخص می‌شود. ولی خب باز هم به خاطر اینکه اجرای تردها غیرقابل پیش‌بینی است، ممکن است همیشه اون خطاها دیده نشوند. یک Uncertainty داریم.
2. Mock و Stub: اگه کدی که تست می‌کنیم به کامپوننت‌های خارجی بستگی داره، می‌توانیم از mock یا stub استفاده کنیم تا رفتار آن بخش‌ها را کنترل کنیم و تست بهتری برای چندین ترد داشته باشیم.
3. ابزارهای تست همزمانی: می‌توانیم از ابزارهایی که برای تست‌های چند تردی و استرس‌تست استفاده می‌شوند، بهره ببریم تا ببینیم توی بار واقعی هم کد درست کار می‌کند یا نه.

سوال سوم - معایب چاپ نتایج آزمون واحد به جای استفاده از assert

تست باید خودکار باشد و بدون دخالت انسانی بتواند تشخیص دهد که آیا یک تست موفق بوده یا شکست خورده است. چاپ نتیجه در کنسول این هدف را نادیده می‌گیرد زیرا نیازمند بررسی دستی توسط خود برنامه نویس است. برای چک کردن نتیجه باید دستی خروجی رو ببینیم و خودمان تصمیم بگیریم که تست پاس شده یا نه. اما وقتی از Assert استفاده می‌کنیم، تست به صورت اتوماتیک ارزیابی می‌شود و نتیجه موفقیت یا شکست رو به شکل خودکار گزارش می‌دهد.

استفاده از چاپ در کنسول گزارش ساختاریافته‌ای ارائه نمی‌دهد که بتوان به‌راحتی مشخص کرد آیا تست موفق یا ناموفق بوده است. این مسئله باعث می‌شود ابزارهایی مانند IDE ها نتوانند نتایج تست را تشخیص دهند و گزارشی در اختیار ما قرار دهند. فرض کنید تعداد زیادی تست داریم، برای اینکه ببینیم کدامشان شکست خورده باید همه‌ی خروجی‌های کنسول را بخوانیم. ولی با Assert اگه تستی شکست بخورد، مستقیما نشون داده می‌شود که کدام بخش ایراد داشته.

استفاده از عبارت assert این امکان را فراهم می‌کند که تست به محض شکست متوقف شده و پیام خطای دقیق ارائه دهد. اما چاپ در کنسول این ویژگی را ندارد و باید خودمان خروجی‌ها را بررسی کنیم تا متوجه شویم که تست در کجا مشکل دارد.

یکی از اهداف تست خودکار این است که به عنوان documentation نیز عمل کند. عبارت‌های assert به‌طور دقیق و شفاف مشخص می‌کنند که نتیجه مورد انتظار چه بوده است و نتیجه واقعی چیست. اما چاپ در کنسول فقط نتیجه را نمایش می‌دهد بدون این‌که توضیح دقیق یا مستندی از انتظار سیستم ارائه دهد. استفاده از System.out.println فقط خروجی رو نشان می‌دهد ولی اطلاعات دقیق‌تری درباره علت شکست ارائه نمی‌دهد. ولی Assert دقیقاً می‌گوید که چه چیزی اشتباه بوده و چرا، مثلا مقادیر ورودی و خروجی رو نشان می‌دهد و می‌گوید تست چه انتظاری داشته.

در صورتی که سیستم تکامل یابد و کد تغییر کند، تست‌ها باید همچنان بدون نیاز به تغییر زیاد، قابل اجرا باشند. چاپ در کنسول می‌تواند موجب شود که developer نتایج تست را به درستی نبیند یا اشتباهاً آن‌ها را نادیده بگیرد.

در ابزارهای خودکارسازی تست خروجی کنسول برای ارزیابی تست‌ها کافی نیست. ابزارهای CI/CD به Assert نیاز دارند تا بتوانند به صورت خودکار بفهمند که تست‌ها موفق بودن یا نه.

سوال چهار - اصلاح تست‌ها

الف) ساختار نادرست برای بررسی Exception: در JUnit ، اگر انتظار داریم که متدی استثنا بدهد، بهتر است از ویژگی‌هایی مثل @Test(expected = Exception.class) یا ویژگی AssertThrows در JUnit استفاده بشود.
این کار، نیاز به بلوک‌های try/catch رو از بین می‌برد و کد تست را تمیزتر و خواناتر می‌کند
نام تست باید Descriptive باشد و کاری که تست انجام می‌دهد را مشخص کند. این کار برای test as documentation ضروری است.
نوع exception ای که باید throw شود، به صورت دقیق مشخص نشده است و نمی‌توان نوع exception ای که از سیستم انتظار می رفت را verify کرد. بهتر است که رفتار سیستم در یک تست به‌صورت کامل و دقیق مشخص و مکتوب شود.
نام متغیر در مورد اینکه چرا این ورودی نامعتبر است، هیچ توضیحی نمی‌دهد و از روی تست نمی‌توان منطق کد را درک کرد. برای مثال، اگر تابع مورد نظر فقط ورودی‌های بزرگ‌تر از صفر را می‌پذیرد، می‌توان ورودی‌های نامعتبر را به‌صورت parameterized تست کرد. برای مثال، یک بار ورودی صفر و یک بار ورودی منفی را آزمایش کرد.
در زیر دو نمونه از اصلاح های پیشنهادی آورده شده است:

```
@Test
public void shouldThrowExceptionForFractionalInput() {
    int invalidFractionalInput = 0.5;

    Exception exception = assertThrows(InvalidInputException.class, () ->
{
        new AnotherClass().process(invalidInput);
    });

    assertEquals("Input must be a positive integer",
exception.getMessage());
}

@ParameterizedTest
@ValueSource(ints = {0, -1, -10})
```

```
public void shouldThrowExceptionForNonPositiveInputs(int invalidInput) {
    Exception exception = assertThrows(InvalidInputException.class, () ->
{
        new AnotherClass().process(invalidInput);
    });

    assertEquals("Input must be a positive integer",
exception.getMessage());
}


@RunWith(Parameterized.class)
@Parameterized.Parameters
public static Object[] data() {
    return new Object[] { 0, -1, -10 };
}

@Test(expected = InvalidInputException.class)
public void shouldThrowExceptionForNonPositiveInputs(int invalidInput) {
    new AnotherClass().process(invalidInput);
}
```

ب) برای تعریف یک class نباید جلوی اسم آن () بذاریم.
هر تست در این کلاس به استیت calculator بستگی دارد. تست دوم به نتیجه تست اول وابسته است و این اصل مستقل بودن تست ها را نقض می کند و باعث ایجاد مشکل می شود. در واقع کد فعلی یک Fixture مشترک رو برای تست‌های مختلف استفاده می‌کند. fixture در هر دو تست به صورت مشترک استفاده می‌شود که باعث می‌شود نتیجه تست‌ها به ترتیب اجرای اون‌ها وابسته باشند. این موضوع می‌تواند باعث بروز تست‌های وابسته به هم (Interacting Tests) بشود. مثلاً، نتیجه‌ی testAccumulate روی testSubsequentAccumulate تاثیر بگذارد. چون در تست دوم مقدار fixture همان مقداری است که از تست اول باقی مانده.
مشکل سوم هم این است که در کد اصلی، چون Fixture مشترک است، اگر یکی از تست‌ها اول اجرا نشود یا شکست بخورد، ممکن است تست های بعدی هم نادرست عمل کنند. این مسئله باعث تست‌های غیرمستقل (Non-Independent Tests) می‌شود.

بهتر است که یک تابع SetUp داشته باشیم و قبل از هر تست استیت calculator را ریست کنیم. همچنین تست‌ها باید کاملاً مستقل از هم باشند و نباید به ترتیب اجرای تست‌ها وابسته باشند. با اطمینان از اینکه هر تست از یک شیء جدید استفاده می‌کند، مشکل سوم هم برطرف می‌شود.
با توجه به اینکه یک instance از کلاس calculator داریم، برای جلوگیری از مشکلات و مستقل کردن تست ها می توان توابع setUp یا tearDown تعریف کرد.
همچنین، بهتر است مقادیر 50، 100 و 150 به صورت متغیر داده شوند تا در صورت عوض کردن عدد، این تغییر propagate نشود و نیاز به تغییر حداقلی داشته باشیم.

```
public class TestCalculator {

    Calculator fixture;

    final int INITIAL_VALUE = 0;

    @BeforeEach
    public void setUp() {
        fixture = Calculator.getInstance();
        fixture.setInitialValue(INITIAL_VALUE);
    }

    @Test
    public void testAccumulate() {
        int toBeAccumulated = 50;
        int expected = INITIAL_VALUE + toBeAccumulated;
        int result = fixture.accumulate(toBeAccumulated);
        Assertions.assertEquals(expected, result);
    }

    @Test
    public void testSubsequentAccumulate() {
        int firstAccumulate = 50;
        int secondAccumulate = 100;
        int expected = firstAccumulate + secondAccumulate;
        fixture.accumulate(firstAccumulate);
        int result = fixture.accumulate(secondAccumulate);
        Assertions.assertEquals(expected, result);
    }
}
```